W3C

REC-html40-19980424

# HTML 4.0 Specification

**W3C Recommendation, revised on *24-Apr-1998***

This version:
    http://www.w3.org/TR/1998/REC-html40-19980424
Latest version:
    http://www.w3.org/TR/REC-html40
Previous version:
    http://www.w3.org/TR/REC-html40-971218
Editors:
    Dave Raggett <dsr@w3.org>
    Arnaud Le Hors <lehors@w3.org>
    Ian Jacobs <ij@w3.org>

## Abstract

This specification defines the HyperText Markup Language (HTML), version 4.0, the publishing language of the World Wide Web. In addition to the text, multimedia, and hyperlink features of the previous versions of HTML, HTML 4.0 supports more multimedia options, scripting languages, style sheets, better printing facilities, and documents that are more accessible to users with disabilities. HTML 4.0 also takes great strides towards the internationalization of documents, with the goal of making the Web truly World Wide.

HTML 4.0 is an SGML application conforming to International Standard ISO 8879 -- Standard Generalized Markup Language [ISO8879].

## Status of this document

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

W3C recommends that user agents and authors (and in particular, authoring tools) produce HTML 4.0 documents rather than HTML 3.2 documents (see [HTML32]). For reasons of backwards compatibility, W3C also recommends that tools interpreting HTML 4.0 continue to support HTML 3.2 and HTML 2.0 as well.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

Public discussion on HTML features takes place on www-html@w3.org.

*This document is a revised version of the document first released on 18 December 1997. Changes from the original version are only editorial in nature.*

# 17 Forms

**Contents**

## 17.1 Introduction to forms

An HTML form is a section of a document containing normal content, markup, special elements called *controls* (checkboxes, radio buttons, menus, etc.), and labels on those controls. Users generally "complete" a form by modifying its controls (entering text selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a Web server, to a mail server, etc.)

Here's a simple form that includes labels, radio buttons, and push buttons (reset the form or submit it):

```
<FORM action="http://somesite.com/prog/adduser" method="post">
    <P>
    <LABEL for="firstname">First name: </LABEL>
              <INPUT type="text" id="firstname"><BR>
    <LABEL for="lastname">Last name: </LABEL>
              <INPUT type="text" id="lastname"><BR>
    <LABEL for="email">email: </LABEL>
              <INPUT type="text" id="email"><BR>
    <INPUT type="radio" name="sex" value="Male"> Male<BR>
    <INPUT type="radio" name="sex" value="Female"> Female<BR>
    <INPUT type="submit" value="Send"> <INPUT type="reset">
    </P>
</FORM>
```

*Note. This specification includes more detailed information about forms in the subsections on form display issues.*

## 17.2 Controls

Users interact with forms through named *controls*.

A control's *"control name"* is given by its `name` attribute. The scope of the `name` attribute for a control within a `FORM` element is the `FORM` element.

Each control has both an initial value and a current value, both of which are character strings. Please consult the definition of each control for information about initial values and possible constraints on values imposed by the control In general, a control's *"initial value"* may be specified with the control element's `value` attribute. However, the initial value of a `TEXTAREA` element is given by its contents, and the initial value of an `OBJECT` element in a form is determine by the object implementation (i.e., it lies outside the scope of this specification).

The control's *"current value"* is first set to the initial value. Thereafter, the control's current value may be modified through user interaction and scripts.

A control's initial value does not change. Thus, when a form is reset, each control's current value is reset to its initial value. If a control does not have an initial value, the effect of a form reset on that control is undefined.

When a form is submitted for processing, some controls have their name paired with their current value and these pairs are submitted with the form. Those controls for which name/value pairs are submitted are called successful controls.

### 17.2.1 Control types

HTML defines the following control types:

**buttons**

Authors may create three types of buttons:

- submit buttons: When activated, a submit button submits a form. A form may contain more than one submit button.
- reset buttons: When activated, a reset button resets all controls to their initial values.
- push buttons: Push buttons have no default behavior. Each push button may have client-side scripts associated with the element's event attributes. When an event occurs (e.g., the user presses the button, releases it, etc.), the associated script is triggered.

  Authors should specify the scripting language of a push button script through a default script declaration (with the `META` element).

Authors create buttons with the BUTTON element or the INPUT element. Please consult the definitions of these elements for details about specifying different button types.

> *Note. Authors should note that the* BUTTON *element offers richer rendering capabilities than the* INPUT *element.*

**checkboxes**
Checkboxes (and radio buttons) are on/off switches that may be toggled by the user. A switch is "on" when the control element's `selected` attribute is set.

When a form is submitted, only "on" checkbox controls can become successful. Several checkboxes in a form may share the same control name. Thus, for example, checkboxes allow users to select several values for the same property. The INPUT element is used to create a checkbox control.

**radio buttons**
Radio buttons are like checkboxes except that when several share the same control name, they are mutually exclusive: when one is switched "on", all others with the same name are switched "off". The INPUT element is used to create a radio button control.

**menus**
Menus offer users options from which to choose. The SELECT element creates a menu, in combination with the OPTGROUP and OPTION elements.

**text input**
Authors may create two types of controls that allow users to input text. The INPUT element creates a single-line input control and the TEXTAREA element creates a multi-line input control. In both cases, the input text becomes the control's current value.

**file select**
This control type allows the user to select files so that their contents may be submitted with a form. The INPUT element is used to create a file select control.

**hidden controls**
Authors may create controls that are not rendered but whose values are submitted with a form. Authors generally use this control type to store information between client/server exchanges that would otherwise be lost due to the stateless nature of HTTP (see [RFC2068]). The INPUT element is used to create a hidden control.

**object controls**
Authors may insert generic objects in forms such that associated values are submitted along with other controls. Authors create object controls with the OBJECT element.

The elements used to create controls generally appear inside a FORM element, but may also appear outside of a FORM element declaration when they are used to build user interfaces. This is discussed in the section on intrinsic events. Note that controls outside a form cannot be successful controls.

## 17.3 The FORM element

```
<!ELEMENT FORM - - (%block;|SCRIPT)+ -(FORM) -- interactive form -->
<!ATTLIST FORM
  %attrs;                              -- %coreattrs, %i18n, %events --
  action      %URI;          #REQUIRED -- server-side form handler --
  method      (GET|POST)     GET       -- HTTP method used to submit the form--
  enctype     %ContentType;  "application/x-www-form-urlencoded"
  onsubmit    %Script;       #IMPLIED  -- the form was submitted --
  onreset     %Script;       #IMPLIED  -- the form was reset --
  accept-charset %Charsets;  #IMPLIED  -- list of supported charsets --
  >
```

*Start tag:* ***required****, End tag:* ***required***

*Attribute definitions*

`action` = *uri* [CT]
: This attribute specifies a form processing agent. For example, the value might be a HTTP URI (to submit the form to a program) or a mailto URI (to email the form).

`method` = `get|post` [CI]
: This attribute specifies which HTTP method will be used to submit the form data set. Possible (case-insensitive) values are "get" (the default) and "post". See the section on form submission for usage information.

`enctype` = *content-type* [CI]
: This attribute specifies the content type used to submit the form to the server (when the value of `method` is "post"). The default value for this attribute is "application/x-www-form-urlencoded". The value "multipart/form-data" should be used in combination with the `INPUT` element, `type`="file".

`accept-charset` = *charset list* [CI]
: This attribute specifies the list of character encodings for input data that must be accepted by the server processing this form. The value is a space- and/or comma-delimited list of charset values. The server must interpret this list as an exclusive-or list, i.e., the server must be able to accept any single character encoding per entity received.

The default value for this attribute is the reserved string "UNKNOWN". User agents may interpret this value as the character encoding that was used to transmit the document containing this `FORM` element.

`accept` = *content-type-list* [CI]
: This attribute specifies a comma-separated list of content types that a server processing this form will handle correctly. User agents may use this information to filter out non-conforming files when prompting a user to selec files to be sent to the server (cf. the `INPUT` element when `type`="file").

*Attributes defined elsewhere*

- `id`, `class` (document-wide identifiers)
- `lang` (language information), `dir` (text direction)
- `style` (inline style information)
- `title` (element title)
- `target` (target frame information)
- `onsubmit`, `onreset`, `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeydown`, `onkeyup` (intrinsic events)

The `FORM` element acts as a container for controls. It specifies:

- The layout of the form (given by the contents of the element).
- The program that will handle the completed and submitted form (the `action` attribute). The receiving program must be able to parse name/value pairs in order to make use of them.
- The method by which user data will be sent to the server (the `method` attribute).
- A character encoding that must be accepted by the server in order to handle this form (the `accept-charset` attribute). User agents may advise the user of the value of the `accept-charset` attribute and/or restrict the user's ability to enter unrecognized characters.

A form can contain text and markup (paragraphs, lists, etc.) in addition to form controls.

The following example shows a form that is to be processed by the "adduser" program when submitted. The form will be sent to the program using the HTTP "post" method.

```
<FORM action="http://somesite.com/prog/adduser" method="post">
...form contents...
```

```
</FORM>
```

The next example shows how to send a submitted form to an email address:

```
<FORM action="mailto:Kligor.T@gee.whiz.com" method="post">
...form contents...
</FORM>
```

Please consult the section on form submission for information about how user agents must prepare form data for servers and how user agents should handle expected responses.

*Note. Further discussion on the behavior of servers that receive form data is beyond the scope of this specification.*

## 17.4 The INPUT element

```
<!ENTITY % InputType
  "(TEXT | PASSWORD | CHECKBOX |
    RADIO | SUBMIT | RESET |
    FILE | HIDDEN | IMAGE | BUTTON)"
   >

<!-- attribute name required for all but submit & reset -->
<!ELEMENT INPUT - O EMPTY              -- form control -->
<!ATTLIST INPUT
  %attrs;                              -- %coreattrs, %i18n, %events --
  type        %InputType;    TEXT      -- what kind of widget is needed --
  name        CDATA          #IMPLIED  -- submit as part of form --
  value       CDATA          #IMPLIED  -- required for radio and checkboxes --
  checked     (checked)      #IMPLIED  -- for radio buttons and check boxes --
  disabled    (disabled)     #IMPLIED  -- unavailable in this context --
  readonly    (readonly)     #IMPLIED  -- for text and passwd --
  size        CDATA          #IMPLIED  -- specific to each type of field --
  maxlength   NUMBER         #IMPLIED  -- max chars for text fields --
  src         %URI;          #IMPLIED  -- for fields with images --
  alt         CDATA          #IMPLIED  -- short description --
  usemap      %URI;          #IMPLIED  -- use client-side image map --
  tabindex    NUMBER         #IMPLIED  -- position in tabbing order --
  accesskey   %Character;    #IMPLIED  -- accessibility key character --
  onfocus     %Script;       #IMPLIED  -- the element got the focus --
  onblur      %Script;       #IMPLIED  -- the element lost the focus --
  onselect    %Script;       #IMPLIED  -- some text was selected --
  onchange    %Script;       #IMPLIED  -- the element value was changed --
  accept      %ContentTypes; #IMPLIED  -- list of MIME types for file upload --
  >
```

*Start tag: **required**, End tag: **forbidden***

*Attribute definitions*

`type` = `text|password|checkbox|radio|submit|reset|file|hidden|image|button` [CI]
    This attribute specifies the type of control to create. The default value for this attribute is "text".
`name` = *cdata* [CI]
    This attribute assigns the control name.
`value` = *cdata* [CA]
    This attribute specifies the initial value of the control. It is optional except when the `type` attribute has the value "radio".
`size` = *cdata* [CN]

This attribute tells the user agent the initial width of the control. The width is given in pixels except when `type` attribute has the value "text" or "password". In that case, its value refers to the (integer) number of characters.

`maxlength` = *number* [CN]
: When the `type` attribute has the value "text" or "password", this attribute specifies the maximum number of characters the user may enter. This number may exceed the specified `size`, in which case the user agent should offer a scrolling mechanism. The default value for this attribute is an unlimited number.

`checked` [CI]
: When the `type` attribute has the value "radio" or "checkbox", this boolean attribute specifies that the button is on User agents must ignore this attribute for other control types.

`src` = *uri* [CT]
: When the `type` attribute has the value "image", this attribute specifies the location of the image to be used to decorate the graphical submit button.

*Attributes defined elsewhere*

- `id`, `class` (document-wide identifiers)
- `lang` (language information), `dir` (text direction)
- `title` (element title)
- `style` (inline style information)
- `alt` (alternate text)
- `align` (alignment)
- `accept` (legal content types for a server)
- `readonly` (read-only input controls)
- `disabled` (disabled input controls)
- `tabindex` (tabbing navigation)
- `accesskey` (access keys)
- `usemap` (client-side image maps)
- `onfocus`, `onblur`, `onselect`, `onchange`, `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeydown`, `onkeyup` (intrinsic events)

## 17.4.1 Control types created with INPUT

The control type defined by the `INPUT` element depends on the value of the `type` attribute:

**text**
: Creates a single-line text input control.

**password**
: Like "text", but the input text is rendered in such a way as to hide the characters (e.g., a series of asterisks). This control type is often used for sensitive input such as passwords. Note that the current value is the text *entered* by the user, not the text rendered by the user agent.

*Note. Application designers should note that this mechanism affords only light security protection. Although the password is masked by user agents from casual observers, it is transmitted to the server in clear text, and may be read by anyone with low-level access to the network.*

**checkbox**
: Creates a checkbox.

**radio**
: Creates a radio button.

**submit**
: Creates a submit button.

**image**

Creates a graphical submit button. The value of the `src` attribute specifies the URI of the image that will decorat the button. For accessibility reasons, authors should provide alternate text for the image via the `alt` attribute.

When a pointing device is used to click on the image, the form is submitted and the click coordinates passed to the server. The x value is measured in pixels from the left of the image, and the y value in pixels from the top of the image. The submitted data includes *name*.x=*x-value* and *name*.y=*y-value* where *"name"* is the value of the `name` attribute, and *x-value* and *y-value* are the x and y coordinate values, respectively.

If the server takes different actions depending on the location clicked, users of non-graphical browsers will be disadvantaged. For this reason, authors should consider alternate approaches:

- Use multiple submit buttons (each with its own image) in place of a single graphical submit button. Authors may use style sheets to control the positioning of these buttons.
- Use a client-side image map together with scripting.

**reset**
Creates a reset button.

**button**
Creates a push button. User agents should use the value of the `value` attribute as the button's label.

**hidden**
Creates a hidden control.

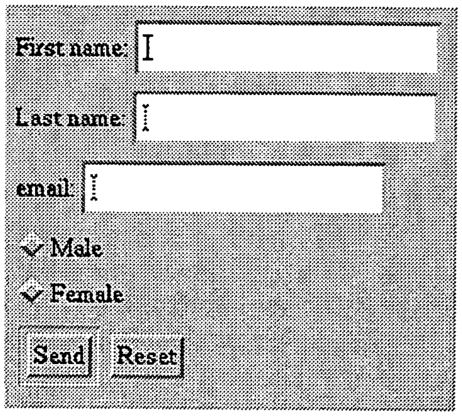
**file**
Creates a file select control. User agents may use the value of the `value` attribute as the initial file name.

## 17.4.2 Examples of forms containing INPUT controls

The following sample HTML fragment defines a simple form that allows the user to enter a first name, last name, emai address, and gender. When the submit button is activated, the form will be sent to the program specified by the `action` attribute.

```
<FORM action="http://somesite.com/prog/adduser" method="post">
   <P>
   First name: <INPUT type="text" name="firstname"><BR>
   Last name: <INPUT type="text" name="lastname"><BR>
   email: <INPUT type="text" name="email"><BR>
   <INPUT type="radio" name="sex" value="Male"> Male<BR>
   <INPUT type="radio" name="sex" value="Female"> Female<BR>
   <INPUT type="submit" value="Send"> <INPUT type="reset">
   </P>
</FORM>
```

This form might be rendered as follows:



In the section on the LABEL element, we discuss marking up labels such as "First name".
In this next example, the JavaScript function name `verify` is triggered when the "onclick" event occurs:

```
<HEAD>
<META http-equiv="Content-Script-Type" content="text/javascript">
</HEAD>
<BODY>
 <FORM action="..." method="post">
    <P>
    <INPUT type="button" value="Click Me" onclick="verify()">
 </FORM>
</BODY>
```

Please consult the section on intrinsic events for more information about scripting and events.
The following example shows how the contents of a user-specified file may be submitted with a form. The user is prompted for his or her name and a list of file names whose contents should be submitted with the form. By specifying the `enctype` value of "multipart/form-data", each file's contents will be packaged for submission in a separate section of a multipart document.

```
<FORM action="http://server.dom/cgi/handle"
    enctype="multipart/form-data"
    method="post">
 <P>
 What is your name? <INPUT type="text" name="name_of_sender">
 What files are you sending? <INPUT type="file" name="name_of_files">
 </P>
</FORM>
```

## 17.5 The BUTTON element

```
<!ELEMENT BUTTON - -
     (%flow;)* -(A|%formctrl;|FORM|FIELDSET)
     -- push button -->
<!ATTLIST BUTTON
  %attrs;                                -- %coreattrs, %i18n, %events --
  name        CDATA          #IMPLIED
  value       CDATA          #IMPLIED  -- sent to server when submitted --
  type        (button|submit|reset) submit -- for use as form button --
  disabled    (disabled)     #IMPLIED  -- unavailable in this context --
  tabindex    NUMBER         #IMPLIED  -- position in tabbing order --
  accesskey   %Character;    #IMPLIED  -- accessibility key character --
  onfocus     %Script;       #IMPLIED  -- the element got the focus --
  onblur      %Script;       #IMPLIED  -- the element lost the focus --
  >
```

*Start tag:* ***required****, End tag:* ***required***

*Attribute definitions*

`name` = *cdata* [CI]
This attribute assigns the control name.

`value` = *cdata* [CS]
This attribute assigns the initial value to the button.

`type` = `submit|button|reset` [CI]
This attribute declares the type of the button. Possible values:

- `submit`: Creates a submit button. This is the default value.
- `reset`: Creates a reset button.

- button: Creates a push button.

*Attributes defined elsewhere*

- id, class (document-wide identifiers)
- lang (language information), dir (text direction)
- title (element title)
- style (inline style information)
- disabled (disabled input controls)
- accesskey (access keys)
- tabindex (tabbing navigation)
- onfocus, onblur, onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (intrinsic events)

Buttons created with the BUTTON element function just like buttons created with the INPUT element, but they offer riche rendering possibilities: the BUTTON element may have content. For example, a BUTTON element that contains an image functions like and may resemble an INPUT element whose type is set to "image", but the BUTTON element type allows content.

Visual user agents may render BUTTON buttons with relief and an up/down motion when clicked, while they may render INPUT buttons as a "flat" images.

The following example expands a previous example, but creates submit and reset buttons with BUTTON instead of INPUT. The buttons contain images by way of the IMG element.

```
<FORM action="http://somesite.com/prog/adduser" method="post">
   <P>
   First name: <INPUT type="text" name="firstname"><BR>
   Last name: <INPUT type="text" name="lastname"><BR>
   email: <INPUT type="text" name="email"><BR>
   <INPUT type="radio" name="sex" value="Male"> Male<BR>
   <INPUT type="radio" name="sex" value="Female"> Female<BR>
   <BUTTON name="submit" value="submit" type="submit">
   Send<IMG src="/icons/wow.gif" alt="wow"></BUTTON>
   <BUTTON name="reset" type="reset">
   Reset<IMG src="/icons/oops.gif" alt="oops"></BUTTON>
   </P>
</FORM>
```

Recall that authors must provide alternate text for an IMG element.

It is illegal to associate an image map with an IMG that appears as the contents of a BUTTON element.

ILLEGAL EXAMPLE:
The following is not legal HTML.

```
<BUTTON>
<IMG src="foo.gif" usemap="...">
</BUTTON>
```

## 17.6 The SELECT, OPTGROUP, and OPTION elements

```
<!ELEMENT SELECT - - (OPTGROUP|OPTION)+ -- option selector -->
<!ATTLIST SELECT
```

```
  %attrs;                                     -- %coreattrs, %i18n, %events --
  name        CDATA          #IMPLIED  -- field name --
  size        NUMBER         #IMPLIED  -- rows visible --
  multiple    (multiple)     #IMPLIED  -- default is single selection --
  disabled    (disabled)     #IMPLIED  -- unavailable in this context --
  tabindex    NUMBER         #IMPLIED  -- position in tabbing order --
  onfocus     %Script;       #IMPLIED  -- the element got the focus --
  onblur      %Script;       #IMPLIED  -- the element lost the focus --
  onchange    %Script;       #IMPLIED  -- the element value was changed --
  >
```

*Start tag: **required**, End tag: **required***

*SELECT Attribute definitions*

name = *cdata* [CI]
: This attribute assigns the control name.

size = *number* [CN]
: If a SELECT element is presented as a scrolled list box, this attribute specifies the number of rows in the list that should be visible at the same time. Visual user agents are not required to present a SELECT element as a list box; they may use any other mechanism, such as a drop-down menu.

multiple [CI]
: If set, this boolean attribute allows multiple selections. If not set, the SELECT element only permits single selections.

*Attributes defined elsewhere*

- id, class (document-wide identifiers)
- lang (language information), dir (text direction)
- title (element title)
- style (inline style information)
- disabled (disabled input controls)
- tabindex (tabbing navigation)
- onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (intrinsic events)

The SELECT element creates a menu. Each choice offered by the menu is represented by an OPTION element. A SELECT element must contain at least one OPTION element.

The OPTGROUP element allows authors to group choices logically. This is particularly helpful when the user must choose from a long list of options; groups of related choices are easier to grasp and remember than a single long list of options. In HTML 4.0, all OPTGROUP elements must be specified directly within a SELECT element (i.e., groups may not be nested).

## 17.6.1 Preselected options

Zero or more choices may be pre-selected for the user. User agents should determine which choices are pre-selected as follows:

- If no OPTION element has the selected attribute set, no options should be pre-selected.
- If one OPTION element has the selected attribute set, it should be pre-selected.
- If the SELECT element has the multiple attribute set and more than one OPTION element has the selected attribute set, they should all be pre-selected.
- It is considered an error if more than one OPTION element has the selected attribute set and the SELECT element

does not have the multiple attribute set. User agents may vary in how they handle this error, but should not pre-select more than one choice.

```
<!ELEMENT OPTGROUP - - (OPTION)+ -- option group -->
<!ATTLIST OPTGROUP
  %attrs;                                -- %coreattrs, %i18n, %events --
  disabled    (disabled)     #IMPLIED  -- unavailable in this context --
  label       %Text;         #REQUIRED -- for use in hierarchical menus --
  >
```

*Start tag: **required**, End tag: **required***

*OPTGROUP Attribute definitions*

label = *text* [CS]
This attribute specifies the label for the option group.

*Attributes defined elsewhere*

- id, class (document-wide identifiers)
- lang (language information), dir (text direction)
- title (element title)
- style (inline style information)
- disabled (disabled input controls)
- onfocus, onblur, onchange, onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (intrinsic events)

*Note. Implementors are advised that future versions of HTML may extend the grouping mechanism to allow for nested groups (i.e., OPTGROUP elements may nest). This will allow authors to represent a richer hierarchy of choices.*

```
<!ELEMENT OPTION - O (#PCDATA)         -- selectable choice -->
<!ATTLIST OPTION
  %attrs;                              -- %coreattrs, %i18n, %events --
  selected    (selected)     #IMPLIED
  disabled    (disabled)     #IMPLIED  -- unavailable in this context --
  label       %Text;         #IMPLIED  -- for use in hierarchical menus --
  value       CDATA          #IMPLIED  -- defaults to element content --
  >
```

*Start tag: **required**, End tag: **optional***

*OPTION Attribute definitions*

selected [CI]
When set, this boolean attribute specifies that this option is pre-selected.
value = *cdata* [CS]
This attribute specifies the initial value of the control. If this attribute is not set, the initial value is set to the contents of the OPTION element.
label = *text* [CS]
This attribute allows authors to specify a shorter label for an option than the content of the OPTION element. When specified, user agents should use the value of this attribute rather than the content of the OPTION element a the option label.

*Attributes defined elsewhere*

- id, class (document-wide identifiers)
- lang (language information), dir (text direction)
- title (element title)
- style (inline style information)
- disabled (disabled input controls)
- onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (intrinsic events)

When rendering a menu choice, user agents should use the value of the label attribute of the OPTION element as the choice. If this attribute is not specified, user agents should use the contents of the OPTION element.

The label attribute of the OPTGROUP element specifies the label for a group of choices.

In this example, we create a menu that allows the user to select which of seven software components to install. The firs and second components are pre-selected but may be deselected by the user. The remaining components are not pre-selected. The size attribute states that the menu should only have 4 rows even though the user may select from among 7 options. The other options should be made available through a scrolling mechanism.

The SELECT is followed by submit and reset buttons.

```
<FORM action="http://somesite.com/prog/component-select" method="post">
   <P>
   <SELECT multiple size="4" name="component-select">
      <OPTION selected value="Component_1_a">Component_1</OPTION>
      <OPTION selected value="Component_1_b">Component_2</OPTION>
      <OPTION>Component_3</OPTION>
      <OPTION>Component_4</OPTION>
      <OPTION>Component_5</OPTION>
      <OPTION>Component_6</OPTION>
      <OPTION>Component_7</OPTION>
   </SELECT>
   <INPUT type="submit" value="Send"><INPUT type="reset">
   </P>
</FORM>
```

Only selected options will be successful (using the control name "component-select"). Note that where the value attribute is set, it determines the control's initial value, otherwise it's the element's contents.
In this example we use the OPTGROUP element to group choices. The following markup:

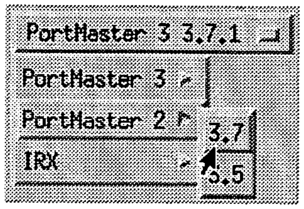
```
<FORM action="http://somesite.com/prog/someprog" method="post">
 <P>
 <SELECT name="ComOS">
     <OPTGROUP label="PortMaster 3">
       <OPTION label="3.7.1" value="pm3_3.7.1">PortMaster 3 with ComOS 3.7.1
       <OPTION label="3.7" value="pm3_3.7">PortMaster 3 with ComOS 3.7
       <OPTION label="3.5" value="pm3_3.5">PortMaster 3 with ComOS 3.5
     </OPTGROUP>
     <OPTGROUP label="PortMaster 2">
       <OPTION label="3.7" value="pm2_3.7">PortMaster 2 with ComOS 3.7
       <OPTION label="3.5" value="pm2_3.5">PortMaster 2 with ComOS 3.5
     </OPTGROUP>
     <OPTGROUP label="IRX">
       <OPTION label="3.7R" value="IRX_3.7R">IRX with ComOS 3.7R
       <OPTION label="3.5R" value="IRX_3.5R">IRX with ComOS 3.5R
     </OPTGROUP>
 </SELECT>
</FORM>
```

represents the following grouping:

```
PortMaster 3
    3.7.1
    3.7
    3.5
PortMaster 2
    3.7
    3.5
IRX
    3.7R
    3.5R
```

Visual user agents may allow users to select from option groups through a hierarchical menu or some other mechanism that reflects the structure of choices.

A graphical user agent might render this as:



This image shows a SELECT element rendered as cascading menus. The top label of the menu displays the currently selected value (PortMaster 3, 3.7.1). The user has unfurled two cascading menus, but has not yet selected the new valu (PortMaster 2, 3.7). Note that each cascading menu displays the label of an OPTGROUP or OPTION element.

## 17.7 The TEXTAREA element

```
<!ELEMENT TEXTAREA - - (#PCDATA)       -- multi-line text field -->
<!ATTLIST TEXTAREA
  %attrs;                              -- %coreattrs, %i18n, %events --
  name        CDATA         #IMPLIED
  rows        NUMBER        #REQUIRED
  cols        NUMBER        #REQUIRED
  disabled    (disabled)    #IMPLIED   -- unavailable in this context --
  readonly    (readonly)    #IMPLIED
  tabindex    NUMBER        #IMPLIED   -- position in tabbing order --
  accesskey   %Character;   #IMPLIED   -- accessibility key character --
  onfocus     %Script;      #IMPLIED   -- the element got the focus --
  onblur      %Script;      #IMPLIED   -- the element lost the focus --
  onselect    %Script;      #IMPLIED   -- some text was selected --
  onchange    %Script;      #IMPLIED   -- the element value was changed --
  >
```

*Start tag:* ***required****, End tag:* ***required***

*Attribute definitions*

name = *cdata* [CI]
: This attribute assigns the control name.

rows = *number* [CN]
: This attribute specifies the number of visible text lines. Users should be able to enter more lines than this, so use agents should provide some means to scroll through the contents of the control when the contents extend beyond

the visible area.

cols = *number* [CN]

This attribute specifies the visible width in average character widths. Users should be able to enter longer lines than this, so user agents should provide some means to scroll through the contents of the control when the contents extend beyond the visible area. User agents may wrap visible text lines to keep long lines visible without the need for scrolling.

*Attributes defined elsewhere*

- id, class (document-wide identifiers)
- lang (language information), dir (text direction)
- title (element title)
- style (inline style information)
- readonly (read-only input controls)
- disabled (disabled input controls)
- tabindex (tabbing navigation)
- onfocus, onblur, onselect, onchange, onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (intrinsic events)

The TEXTAREA element creates a multi-line text input control. User agents should use the contents of this element as the initial value of the control and should render this text initially.

This example creates a TEXTAREA control that is 20 rows by 80 columns and contains two lines of text initially. The TEXTAREA is followed by submit and reset buttons.

```
<FORM action="http://somesite.com/prog/text-read" method="post">
   <P>
   <TEXTAREA name="thetext" rows="20" cols="80">
   First line of initial text.
   Second line of initial text.
   </TEXTAREA>
   <INPUT type="submit" value="Send"><INPUT type="reset">
   </P>
</FORM>
```

Setting the readonly attribute allows authors to display unmodifiable text in a TEXTAREA. This differs from using standard marked-up text in a document because the value of TEXTAREA is submitted with the form.

## 17.8 The ISINDEX element

**ISINDEX is deprecated.** This element creates a single-line text input control. Authors should use the INPUT element t create text input controls.

See the Transitional DTD for the formal definition.

*Attribute definitions*

prompt = *text* [CS]

**Deprecated.** This attribute specifies a prompt string for the input field.

*Attributes defined elsewhere*

- id, class (document-wide identifiers)

- lang (language information), dir (text direction)
- title (element title)
- style (inline style information)

The ISINDEX element creates a single-line text input control that allows any number of characters. User agents may use the value of the prompt attribute as a title for the prompt.

DEPRECATED EXAMPLE:
The following ISINDEX declaration:

```
<ISINDEX prompt="Enter your search phrase: ">
```

could be rewritten with INPUT as follows:

```
<FORM action="..." method="post">
<P>Enter your search phrase: <INPUT type="text"></P>
</FORM>
```

***Semantics of ISINDEX**. Currently, the semantics for ISINDEX are only well-defined when the base URI for the enclosing document is an HTTP URI. In practice, the input string is restricted to Latin-1 as there is no mechanism for the URI to specify a different character set.*

## 17.9 Labels

Some form controls automatically have labels associated with them (press buttons) while most do not (text fields, checkboxes and radio buttons, and menus).

For those controls that have implicit labels, user agents should use the value of the value attribute as the label string.

The LABEL element is used to specify labels for controls that do not have implicit labels,

### 17.9.1 The LABEL element

```
<!ELEMENT LABEL - - (%inline;)* -(LABEL) -- form field label text -->
<!ATTLIST LABEL
  %attrs;                              -- %coreattrs, %i18n, %events --
  for         IDREF        #IMPLIED  -- matches field ID value --
  accesskey   %Character;  #IMPLIED  -- accessibility key character --
  onfocus     %Script;     #IMPLIED  -- the element got the focus --
  onblur      %Script;     #IMPLIED  -- the element lost the focus --
  >
```

*Start tag: **required**, End tag: **required***

*Attribute definitions*

for = *idref* [CS]
: This attribute explicitly associates the label being defined with another control. When present, the value of this attribute must be the same as the value of the id attribute of some other control in the same document. When absent, the label being defined is associated with the element's contents.

*Attributes defined elsewhere*

- id, class (document-wide identifiers)
- lang (language information), dir (text direction)
- title (element title)
- style (inline style information)
- accesskey (access keys)
- onfocus, onblur, onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (intrinsic events)

The LABEL element may be used to attach information to controls. Each LABEL element is associated with exactly one form control.

The for attribute associates a label with another control explicitly: the value of the for attribute must be the same as the value of the id attribute of the associated control element. More than one LABEL may be associated with the same control by creating multiple references via the for attribute.

This example creates a table that is used to align two text input controls and their associated labels. Each label is associated explicitly with one text input:

```
<FORM action="..." method="post">
<TABLE>
  <TR>
    <TD><LABEL for="fname">First Name</LABEL>
    <TD><INPUT type="text" name="firstname" id="fname">
  <TR>
    <TD><LABEL for="lname">Last Name</LABEL>
    <TD><INPUT type="text" name="lastname" id="lname">
</TABLE>
</FORM>
```

This example extends a previous example form to include LABEL elements.

```
 <FORM action="http://somesite.com/prog/adduser" method="post">
    <P>
    <LABEL for="firstname">First name: </LABEL>
              <INPUT type="text" id="firstname"><BR>
    <LABEL for="lastname">Last name: </LABEL>
              <INPUT type="text" id="lastname"><BR>
    <LABEL for="email">email: </LABEL>
              <INPUT type="text" id="email"><BR>
    <INPUT type="radio" name="sex" value="Male"> Male<BR>
    <INPUT type="radio" name="sex" value="Female"> Female<BR>
    <INPUT type="submit" value="Send"> <INPUT type="reset">
    </P>
 </FORM>
```

To associate a label with another control implicitly, the control element must be within the contents of the LABEL element. In this case, the LABEL may only contain one control element. The label itself may be positioned before or after the associated control.

In this example, we implicitly associate two labels with two text input controls:

```
<FORM action="..." method="post">
<P>
<LABEL>
   First Name
   <INPUT type="text" name="firstname">
</LABEL>
<LABEL>
```

```
    <INPUT type="text" name="lastname">
    Last Name
  </LABEL>
  </P>
  </FORM>
```

Note that this technique cannot be used when a table is being used for layout, with the label in one cell and its associated control in another cell.

When a LABEL element receives focus, it passes the focus on to its associated control. See the section below on access keys for examples.

Labels may be rendered by user agents in a number of ways (e.g., visually, read by speech synthesizers, etc.)

## 17.10 Adding structure to forms: the FIELDSET and LEGEND elements

```
<!--
  #PCDATA is to solve the mixed content problem,
  per specification only whitespace is allowed there!
 -->
<!ELEMENT FIELDSET - - (#PCDATA,LEGEND,(%flow;)*) -- form control group -->
<!ATTLIST FIELDSET
  %attrs;                                -- %coreattrs, %i18n, %events --
  >

<!ELEMENT LEGEND - - (%inline;)*       -- fieldset legend -->
<!ENTITY % LAlign "(top|bottom|left|right)">

<!ATTLIST LEGEND
  %attrs;                                -- %coreattrs, %i18n, %events --
  accesskey   %Character;     #IMPLIED   -- accessibility key character --
  >
```

*Start tag:* ***required****, End tag:* ***required***

*LEGEND Attribute definitions*

align = top|bottom|left|right [CI]
**Deprecated.** This attribute specifies the position of the legend with respect to the fieldset. Possible values:
- top: The legend is at the top of the fieldset. This is the default value.
- bottom: The legend is at the bottom of the fieldset.
- left: The legend is at the left side of the fieldset.
- right: The legend is at the right side of the fieldset.

*Attributes defined elsewhere*

- id, class (document-wide identifiers)
- lang (language information), dir (text direction)
- title (element title)
- style (inline style information)
- accesskey (access keys)
- onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (intrinsic events)

The FIELDSET element allows authors to group thematically related controls and labels. Grouping controls makes it easier for users to understand their purpose while simultaneously facilitating tabbing navigation for visual user agents and speech navigation for speech-oriented user agents. The proper use of this element makes documents more accessible.

The LEGEND element allows authors to assign a caption to a FIELDSET. The legend improves accessibility when the FIELDSET is rendered non-visually.

In this example, we create a form that one might fill out at the doctor's office. It is divided into three sections: personal information, medical history, and current medication. Each section contains controls for inputting the appropriate information.

```
<FORM action="..." method="post">
 <P>
 <FIELDSET>
  <LEGEND>Personal Information</LEGEND>
  Last Name: <INPUT name="personal_lastname" type="text" tabindex="1">
  First Name: <INPUT name="personal_firstname" type="text" tabindex="2">
  Address: <INPUT name="personal_address" type="text" tabindex="3">
  ...more personal information...
 </FIELDSET>
 <FIELDSET>
  <LEGEND>Medical History</LEGEND>
  <INPUT name="history_illness"
         type="checkbox"
         value="Smallpox" tabindex="20"> Smallpox
  <INPUT name="history_illness"
         type="checkbox"
         value="Mumps" tabindex="21"> Mumps
  <INPUT name="history_illness"
         type="checkbox"
         value="Dizziness" tabindex="22"> Dizziness
  <INPUT name="history_illness"
         type="checkbox"
         value="Sneezing" tabindex="23"> Sneezing
  ...more medical history...
 </FIELDSET>
 <FIELDSET>
  <LEGEND>Current Medication</LEGEND>
  Are you currently taking any medication?
  <INPUT name="medication_now"
         type="radio"
         value="Yes" tabindex="35">Yes
  <INPUT name="medication_now"
         type="radio"
         value="No" tabindex="35">No

  If you are currently taking medication, please indicate
  it in the space below:
  <TEXTAREA name="current_medication"
            rows="20" cols="50"
            tabindex="40">
  </TEXTAREA>
 </FIELDSET>
</FORM>
```

Note that in this example, we might improve the visual presentation of the form by aligning elements within each FIELDSET (with style sheets), adding color and font information (with style sheets), adding scripting (say, to only open the "current medication" text area if the user indicates he or she is currently on medication), etc.

## 17.11 Giving focus to an element

In an HTML document, an element must receive *focus* from the user in order to become active and perform its tasks. For example, users must activate a link specified by the A element in order to follow the specified link. Similarly, users must give a TEXTAREA focus in order to enter text into it.

There are several ways to give focus to an element:

- Designate the element with a pointing device.
- Navigate from one element to the next with the keyboard. The document's author may define a *tabbing order* tha specifies the order in which elements will receive focus if the user navigates the document with the keyboard (se tabbing navigation). Once selected, an element may be activated by some other key sequence.
- Select an element through an *access key* (sometimes called "keyboard shortcut" or "keyboard accelerator").

### 17.11.1 Tabbing navigation

*Attribute definitions*

`tabindex` = *number* [CN]
: This attribute specifies the position of the current element in the tabbing order for the current document. This value must be a number between 0 and 32767. User agents should ignore leading zeros.

The *tabbing order* defines the order in which elements will receive focus when navigated by the user via the keyboard. The tabbing order may include elements nested within other elements.

Elements that may receive focus should be navigated by user agents according to the following rules:

1. Those elements that support the `tabindex` attribute and assign a positive value to it are navigated first. Navigation proceeds from the element with the lowest `tabindex` value to the element with the highest value. Values need not be sequential nor must they begin with any particular value. Elements that have identical `tabindex` values should be navigated in the order they appear in the character stream.
2. Those elements that do not support the `tabindex` attribute or support it and assign it a value of "0" are navigated next. These elements are navigated in the order they appear in the character stream.
3. Elements that are disabled do not participate in the tabbing order.

The following elements support the `tabindex` attribute: A, AREA, BUTTON, INPUT, OBJECT, SELECT, and TEXTAREA.

In this example, the tabbing order will be the BUTTON, the INPUT elements in order (note that "field1" and the button share the same tabindex, but "field1" appears later in the character stream), and finally the link created by the A element.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
   "http://www.w3.org/TR/REC-html40/strict.dtd">
<HTML>
<HEAD>
<TITLE>A document with FORM</TITLE>
</HEAD>
<BODY>
...some text...
<P>Go to the
<A tabindex="10" href="http://www.w3.org/">W3C Web site.</A>
...some more...
<BUTTON type="button" name="get-database"
           tabindex="1" onclick="get-database">
```

```
Get the current database.
</BUTTON>
...some more...
<FORM action="..." method="post">
<P>
<INPUT tabindex="1" type="text" name="field1">
<INPUT tabindex="2" type="text" name="field2">
<INPUT tabindex="3" type="submit" name="submit">
</P>
</FORM>
</BODY>
</HTML>
```

***Tabbing keys.*** *The actual key sequence that causes tabbing navigation or element activation depends on the configuration of the user agent (e.g., the "tab" key is used for navigation and the "enter" key is used to activate a selected element).*

*User agents may also define key sequences to navigate the tabbing order in reverse. When the end (or beginning) of the tabbing order is reached, user agents may circle back to the beginning (or end).*

### 17.11.2 Access keys

*Attribute definitions*

accesskey = *character* [CN]
This attribute assigns an access key to an element. An access key is a single character from the document character set.**Note.** Authors should consider the input method of the expected reader when specifying an accesskey.

Pressing an access key assigned to an element gives focus to the element. The action that occurs when an element receives focus depends on the element. For example, when a user activates a link defined by the A element, the user agent generally follows the link. When a user activates a radio button, the user agent changes the value of the radio button. When the user activates a text field, it allows input, etc.

The following elements support the accesskey attribute: A, AREA, BUTTON, INPUT, LABEL, and LEGEND, and TEXTAREA.

This example assigns the access key "U" to a label associated with an INPUT control. Typing the access key gives focus to the label which in turn gives it to the associated control. The user may then enter text into the INPUT area.

```
<FORM action="..." method="post">
<P>
<LABEL for="fuser" accesskey="U">
User Name
</LABEL>
<INPUT type="text" name="user" id="fuser">
</P>
</FORM>
```

In this example, we assign an access key to a link defined by the A element. Typing this access key takes the user to another document, in this case, a table of contents.

```
<P><A accesskey="C"
      rel="contents"
      href="http://someplace.com/specification/contents.html">
    Table of Contents</A>
```

The invocation of access keys depends on the underlying system. For instance, on machines running MS Windows, on generally has to press the "alt" key in addition to the access key. On Apple systems, one generally has to press the "cmd" key in addition to the access key.

The rendering of access keys depends on the user agent. We recommend that authors include the access key in label text or wherever the access key is to apply. User agents should render the value of an access key in such a way as to emphasize its role and to distinguish it from other characters (e.g., by underlining it).

## 17.12 Disabled and read-only controls

In contexts where user input is either undesirable or irrelevant, it is important to be able to disable a control or render it read-only. For example, one may want to disable a form's submit button until the user has entered some required data. Similarly, an author may want to include a piece of read-only text that must be submitted as a value along with the form. The following sections describe disabled and read-only controls.

### 17.12.1 Disabled controls

*Attribute definitions*

`disabled` [CI]
    When set for a form control, this boolean attribute disables the control for user input.

When set, the `disabled` attribute has the following effects on an element:

- Disabled controls do not receive focus.
- Disabled controls are skipped in tabbing navigation.
- Disabled controls cannot be successful.

The following elements support the `disabled` attribute: `BUTTON` `INPUT`, `OPTGROUP`, `OPTION`, `SELECT`, and `TEXTAREA`.

This attribute is inherited but local declarations override the inherited value.

How disabled elements are rendered depends on the user agent. For example, some user agents "gray out" disabled menu items, button labels, etc.

In this example, the `INPUT` element is disabled. Therefore, it cannot receive user input nor will its value be submitted with the form.

```
<INPUT disabled name="fred" value="stone">
```

***Note.** The only way to modify dynamically the value of the* `disabled` *attribute is through a script.*

### 17.12.2 Read-only controls

*Attribute definitions*

`readonly` [CI]
    When set for a form control, this boolean attribute prohibits changes to the control.

The `readonly` attribute specifies whether the control may be modified by the user.

When set, the `readonly` attribute has the following effects on an element:

- Read-only elements receive focus but cannot be modified by the user.
- Read-only elements are included in tabbing navigation.
- Read-only elements may be successful.

The following elements support the `readonly` attribute: `INPUT` and `TEXTAREA`.

How read-only elements are rendered depends on the user agent.

*Note. The only way to modify dynamically the value of the* `readonly` *attribute is through a script.*

## 17.13 Form submission

The following sections explain how user agents submit form data to form processing agents.

### 17.13.1 Form submission method

The `method` attribute of the `FORM` element specifies the HTTP method used to send the form to the processing agent. This attribute may take two values:

- `get`: With the HTTP "get" method, the form data set is appended to the URI specified by the `action` attribute (with a question-mark ("?") as separator) and this new URI is sent to the processing agent.
- `post`: With the HTTP "post" method, the form data set is included in the body of the form and sent to the processing agent.

The "get" method should be used when the form is idempotent (i.e., causes no side-effects). Many database searches have no visible side-effects and make ideal applications for the "get" method.

If the service associated with the processing of a form causes side effects (for example, if the form modifies a database or subscription to a service), the "post" method should be used.

*Note. The "get" method restricts form data set values to ASCII characters. Only the "post" method (with* `enctype`*="multipart/form-data") is specified to cover the entire [ISO10646] character set.*

### 17.13.2 Successful controls

A *successful control* is "valid" for submission. Every successful control has its control name paired with its current value as part of the submitted form data set. A successful control must be defined within a `FORM` element and must have a control name.

However:

- Controls that are disabled cannot be successful.
- If a form contains more than one submit button, only the activated submit button is successful.
- All "on" checkboxes may be successful.
- For radio buttons that share the same value of the `name` attribute, only the "on" radio button may be successful.
- For menus, the control name is provided by a `SELECT` element and values are provided by `OPTION` elements. Only selected options may be successful.
- The current value of a file select is a list of one or more file names. Upon submission of the form, the *contents* of each file are submitted with the rest of the form data. The file contents are packaged according to the form's content type.
- The current value of an object control is determined by the object's implementation.

If a control doesn't have a current value when the form is submitted, user agents are not required to treat it as a successful control.

Furthermore, user agents should not consider the following controls successful:

- Reset buttons.
- `OBJECT` elements whose `declare` attribute has been set.

Hidden controls and controls that are not rendered because of style sheet settings may still be successful. For example:

```
<FORM action="..." method="post">
<P>
<INPUT type="password" style="display:none"
          name="invisible-password"
          value="mypassword">
</FORM>
```

will still cause a value to be paired with the name "invisible-password" and submitted with the form.

## 17.13.3 Processing form data

When the user submits a form (e.g., by activating a submit button), the user agent processes it as follows.

**Step one: Identify the successful controls**

**Step two: Build a form data set**

A *form data set* is a sequence of control-name/current-value pairs constructed from successful controls

**Step three: Encode the form data set**

The form data set is then encoded according to the content type specified by the `enctype` attribute of the `FORM` element.

**Step four: Submit the encoded form data set**

Finally, the encoded data is sent to the processing agent designated by the `action` attribute using the protocol specified by the `method` attribute.

This specification does not specify all valid submission methods or content types that may be used with forms. However, HTML 4.0 user agents must support the established conventions in the following cases:

- If the `method` is "get" and the `action` is an HTTP URI, the user agent takes the value of `action`, appends a `?' to it, then appends the form data set, encoded using the "application/x-www-form-urlencoded" content type. The user agent then traverses the link to this URI. In this scenario, form data are restricted to ASCII codes.
- If the `method` is "post" and the `action` is an HTTP URI, the user agent conducts an HTTP "post" transaction using the value of the `action` attribute and a message created according to the content type specified by the `enctype` attribute.

For any other value of `action` or `method`, behavior is unspecified.

User agents should render the response from the HTTP "get" and "post" transactions.

## 17.13.4 Form content types

The enctype attribute of the FORM element specifies the content type used to encode the form data set for submission tc the server. User agents must support the content types listed below. Behavior for other content types is unspecified.

Please also consult the section on escaping ampersands in URI attribute values.

**application/x-www-form-urlencoded**

This is the default content type. Forms submitted with this content type must be encoded as follows:

1. Control names and values are escaped. Space characters are replaced by `+', and then reserved characters are escaped as described in [RFC1738], section 2.2: Non-alphanumeric characters are replaced by `%HH', a percent sign and two hexadecimal digits representing the ASCII code of the character. Line breaks are represented as "CR LF" pairs (i.e., `%0D%0A').
2. The control names/values are listed in the order they appear in the document. The name is separated from the value by `=' and name/value pairs are separated from each other by `&'.

**multipart/form-data**

*Note. Please consult [RFC1867] for additional information about file uploads, including backwards compatibility issues, the relationship between "multipart/form-data" and other content types, performance issues, etc.*

*Please consult the appendix for information about security issues for forms.*

The content type "application/x-www-form-urlencoded" is inefficient for sending large quantities of binary data or text containing non-ASCII characters. The content type "multipart/form-data" should be used for submitting forms that contain files, non-ASCII data, and binary data.

The content "multipart/form-data" follows the rules of all multipart MIME data streams as outlined in [RFC2045]. The definition of "multipart/form-data" is available at the [IANA] registry.

A "multipart/form-data" message contains a series of parts, each representing a successful control. The parts are sent to the processing agent in the same order the corresponding controls appear in the document stream. Part boundaries should not occur in any of the data; how this is done lies outside the scope of this specification.

As with all multipart MIME types, each part has an optional "Content-Type" header that defaults to "text/plain". User agents should supply the "Content-Type" header, accompanied by a "charset" parameter.

Each part is expected to contain:

1. a "Content-Disposition" header whose value is "form-data".
2. a name attribute specifying the control name of the corresponding control. Control names originally encoded in non-ASCII character sets may be encoded using the method outlined in [RFC2045].

Thus, for example, for a control named "mycontrol", the corresponding part would be specified:

```
Content-Disposition: form-data; name="mycontrol"
```

As with all MIME transmissions, "CR LF" (i.e., `%0D%0A') is used to separate lines of data.

Each part may be encoded and the "Content-Transfer-Encoding" header supplied if the value of that part does not conform to the default (7BIT) encoding (see [RFC2045], section 6)

If the contents of a file are submitted with a form, the file input should be identified by the appropriate content type (e.g., "application/octet-stream"). If multiple files are to be returned as the result of a single form entry, they should be returned as "multipart/mixed" embedded within the "multipart/form-data".

The user agent should attempt to supply a file name for each submitted file. The file name may be specified with the "filename" parameter of the 'Content-Disposition: form-data' header, or, in the case of multiple files, in a 'Content-Disposition: file' header of the subpart. If the file name of the client's operating system is not in US-ASCII, the file name might be approximated or encoded using the method of [RFC2045]. This is convenient for those cases where, for example, the uploaded files might contain references to each other (e.g., a TeX file and its ".sty" auxiliary style description).

The following example illustrates "multipart/form-data" encoding. Suppose we have the following form:

```
<FORM action="http://server.dom/cgi/handle"
      enctype="multipart/form-data"
      method="post">
  <P>
  What is your name? <INPUT type="text" name="submit-name"><BR>
  What files are you sending? <INPUT type="file" name="files"><BR>
  <INPUT type="submit" value="Send"> <INPUT type="reset">
</FORM>
```

If the user enters "Larry" in the text input, and selects the text file "file1.txt", the user agent might send back the following data:

```
Content-Type: multipart/form-data; boundary=AaB03x

--AaB03x
Content-Disposition: form-data; name="submit-name"

Larry
--AaB03x
Content-Disposition: form-data; name="files"; filename="file1.txt"
Content-Type: text/plain

... contents of file1.txt ...
--AaB03x--
```

If the user selected a second (image) file "file2.gif", the user agent might construct the parts as follows:

```
Content-Type: multipart/form-data; boundary=AaB03x

--AaB03x
Content-Disposition: form-data; name="submit-name"

Larry
--AaB03x
Content-Disposition: form-data; name="files"
Content-Type: multipart/mixed; boundary=BbC04y

--BbC04y
Content-Disposition: attachment; filename="file1.txt"
Content-Type: text/plain

... contents of file1.txt ...
--BbC04y
Content-Disposition: attachment; filename="file2.gif"
Content-Type: image/gif
Content-Transfer-Encoding: binary
```

```
:..contents of file2.gif...
--BbC04y--
--AaB03x--
```

---